On the effectiveness of application-aware self-management for scientific discovery in volunteer computing systems

Trilce Estrada and Michela Taufer
Department of Computer and Information Science
University of Delaware
Newark, Delaware, 19716
Email: estrada, taufer@udel.edu

***Abstract*—An important challenge faced by high-throughput, multiscale applications is that human intervention has a central role in driving their success. However, manual intervention is inefficient, error-prone and promotes resource wasting. This paper presents an application-aware modular framework that provides self-management for computational multiscale applications in volunteer computing (VC). Our framework consists of a learning engine and three modules that can be easily adapted to different distributed systems. The learning engine of this framework is based on our novel tree-like structure called KOTree. KOTree is a fully automatic method that organizes statistical information in a multidimensional structure that can be efficiently searched and updated at runtime. Our empirical evaluation shows that our framework can effectively provide application-aware self-management in VC systems. Additionally, we observed that our KOTree algorithm is able to predict accurately the expected length of new jobs, resulting in an average of 85% increased throughput with respect to other algorithms.**

I. Introduction

Multiscale modeling and simulation is one of the hardest challenges that science faces today and is becoming a crucial approach to investigate complex problems in applied sciences. Multiscale simulations are usually composed by a sequence of modules that address a problem at different scales. The behavior of a module can be differently affected by one or more parameters. For example, in a module based on Molecular Dynamics (MD) [1], parameters can include initial velocity, number of cooling and heating steps, solvent type, and docking methods. In this example, initial velocity and number of cooling or heating steps affect the accuracy of the solution. Solvent type, depending on whether it is explicit or implicit, can affect accuracy of the solution, size of the problem, or both. This uncertainty becomes even more dramatic when several modules and scales interact with each other, and is due to the complicated interactions between multiple scales and algorithms that are usually inherent of multiscale applications.

An important drawback faced by high-throughput, multiscale applications is the central role of human intervention. Many times scientists must carefully analyze results either manually or semi-automatically and then make the appropriate decisions to tune application parameters. For better resource utilization this analysis and subsequent tuning should be done as soon as new results, possibly millions of them, are collected. However, scientists tuning projects may not be aware of the full impact of different combinations of parameters, resulting in unexpected consequences such as batches of simulations that are extremely expensive or inaccurate. Thus, direct human intervention is not only tiring and highly demanding on the scientist, but costly and detrimental to the efficient use of large amounts of computing resources.

In this paper we claim that high-throughput, multiscale applications need to become self-managed. An autonomic system, or self-managed system, has the capability of configuring, diagnosing, optimizing, and protecting itself [2]. Although most distributed systems already provide some degree of self-management, they do not consider application-specific optimization of parameters. Thus, we propose an autonomic, modular framework for multiscale applications running on distributed systems. In particular, we use volunteer computing (VC) as a representative of a large and heterogeneous distributed system that is in need of application-aware self-management. Our contributions are:

- A novel multidimensional tree-like data structure and its algorithms called Knowledge Organization Tree (KOTree) that organizes statistical information for efficient exploration and prediction of application parameters while being built and updated at runtime.
- A modular framework, integrating KOTree, for providing application-aware self-management in VC that can be easily adapted to different distributed systems and applications with diverse scientific goals.

The organization of this paper is as follows: Section II explores the state of the art in stream mining. Section III defines the parametric nature of VC applications. Section IV describes the structure and functionality of KOTrees. Section V shows implementation aspects of our modular framework. Section VI validates our framework with respect to three application types with diverse application goals. Section VII concludes the paper with a discussion about the effectiveness of our proposed framework and future work.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

II. State of the art

To provide self-management in large and dynamic systems such as VC, we need to analyze large amounts of information at runtime and eventually discard the information to avoid storage pressure. Traditional search strategies such as A^* [3], cannot optimally perform for this problem because the space representation is extremely large. It is also not practical to store and analyze all the returned jobs and even less practical to build a graph of them. Other methods, such as EM [4] and Monte Carlo [5], cannot be applied efficiently because of the large size of the problem, the dynamism of the environment, and its intrinsic delay to explore candidate solutions.

A method capable of providing application-aware self-management in VC needs to meet the following requirements: a model has to be built and updated at runtime; the model needs to be able to make predictions for several application metrics in near real time; the same model has to scale linearly and learn from observed data in a single pass; and the model has to efficiently identify parameter combinations that can advance the VC application goal. To meet these requirements a natural choice is using a stream mining approach, including clustering, lazy learning, neural networks, Bayesian learning, or decision trees. However, as discussed in this section, none of these methods meet all the requirements listed above.

Clustering techniques such as those proposed by Guha *et al.* [6] and Bartlett *et al.* [7] can be potentially used to identifying parameter combinations that are more likely to advance the application goal. However, these clustering techniques are not efficient at updating themselves at runtime under dynamic conditions. Our approach is similar to clustering in the sense that it groups parameters based on their relative 'closeness', but it is different from these approaches in the sense that the clustering in our KOTree algorithm happens in a constant time complexity as a consequence of the data organization that we propose. Additionally, we do not need to store actual jobs or reorganize our data in the KOTree structure when new data are collected.

Lazy learning and *nearest neighbors* algorithms are the easiest family of algorithms to be adapted for stream mining because they adapt automatically when a new job is received and there is no training cost associated to them. However, this type of algorithms do not scale well with massive amounts of data since it is not possible to store the entire dataset. Moreover searching for the nearest neighbor or producing a local model every time a job needs to be classified is expensive. This problem has been addressed and partially mitigated by Zhang *et al.* [8], Yang *et al.* [9], Ueno [10], and Erdogan *et al.* [11], for a variety of applications. However, lazy and nearest neighbor approaches are still affected by storage and memory constraints. Although the problem can be mitigated with hybrid approaches, a lazy approach is still not suitable for systems collecting very large amounts of data such as VC systems.

Neural networks and *Bayesian learning* can be efficiently used for stream mining because training occurs incrementally. Contrary to lazy learning they do not need to store the jobs dataset, thus, avoiding storage pressure. Some real-world examples using this type of learning include He *et al.* [12], Leng *et al.* [13], Raahemi *et al* [14], Kawashima *et al.* [15], Qing *et al.* [16], and Al Machot *et al.* [17]. Neural networks and Bayesian learning are approaches that meet most of the requirements we need for learning from streams in VC systems. However, in addition to fast training and prediction, we also need an effective mechanism for further exploring the parameter space. This feature is not provided by either neural networks or Bayesian learning.

Hoeffding trees currently represent state-of-the-art for classifying high-speed data streams. They were introduced by Domingos and Hulten in [18]. The idea is to use the Hoeffding bounds to grow a tree incrementally. Every job is examined only once and the tree stores statistical information on how to grow and split nodes in the tree. Our KOTree algorithm is an N-$dimensional$ tree inspired by the Hoeffding trees in the sense that each node collects statistical information about the observed jobs. However, our approach differs in that every node represents a collection of N tests applied over N attributes (on a 1 to 1 basis) rather than a single test over one attribute. Thus, a node in our tree represents a hypercube in the N-$dimensional$ space and allows us identifying efficiently parameter combinations that can potentially advance VC application goals. Another key difference is that contrary to Hoeffding trees, KOTree is parametric and has a maximum expansion factor, or height, predefined by the user.

In the field of autonomic management and closely related to our proposed work, Yang *et al.* [19], present an autonomic runtime manager (ARM) that uses the application spatial and temporal characteristics as well as resource status as the main criteria to self-optimize the execution of distributed applications at runtime. Our proposed application-aware self-management VC framework also uses temporal assessment to adapt itself at the time goes by, but the key difference between our work and ARM is the system they optimize; while we focus on a complex VC system, Yang *et al.* optimize a more simplified distributed system.

The closest approach in essence and scope to our approach for providing application-aware self-management in VC, is the work that Moore Jr. *et al.* propose in [20]. This work refers to MindModeling@Home, a VC project for cognitive modeling that is concerned with understanding changes on model performance across non-linear mechanisms and parameters. Similarly to our approach Moore Jr. *et al.* propose the Cell mechanism to explore the parameter space. Contrary to our framework, Cell does not integrate system knowledge, thus Cell has difficulty keeping enough work units ready for distribution. Additionally, the Cell approach does not scale for large amounts of data since it maintains the data in memory for computing regression planes. Our proposed approach based on KOTrees solves the space and memory problem because we do not need to store the results of returned jobs.

III. Parametric applications in VC

Volunteer Computing (VC) is a distributed paradigm where computers connected to Internet donate their idle cycles for scientific computation. BOINC [21] is a well-known middleware that enables VC. In this paper we chose to present our contributions in the context of VC, not because they are tied to the paradigm itself, but because a VC system has the potential of producing millions of results per day, even though execution time of individual jobs in VC is in the order of hours or days. Additionally, scientific applications in VC systems are diverse in scope and objectives, making it harder to come up with a manual solution that can fit them all.

Based on their goals we classify applications as: coverage-, accuracy-, convergence-, and latency-oriented applications. Coverage-oriented applications have great throughput requirements and are usually in the form of a search in a huge space; the more space searched, the better. An example of a coverage-oriented application is Einstein@home [22] that searches for spinning neutron stars, or pulsars, using data from the LIGO and GEO gravitational wave detectors. The goal is exploring as many different regions of the sky as possible.

Accuracy-oriented applications are concerned about the accuracy of their individual results. Accuracy-oriented applications usually fine-tune multiple sets of parameters and accuracy of each single job matters. An example of an accuracy-oriented application is Climateprediction.net [23], which is computing a massive environmental model intended to forecast climate conditions in the 21st century. Its goal is to build a global climate model out of many accurate, partial models.

Convergence-oriented applications are a combination of accuracy- and coverage-oriented applications. Both goals are equally important; these applications need to explore large search spaces and provide, usually, a single accurate global solution. An example of a convergence-oriented application is Docking@home [24], which simulates protein-ligand docking processes. The goal is finding the native-like docking conformation by extensively exploring the very large search space of docking conformations.

Latency-oriented applications primary goal is to reduce time to a solution. An example of a latency-oriented application is the quake-catcher network (QNC) [25] that performs earthquake monitoring. The goal is providing a quick response in the event of an earthquake.

Thus, high-throughput applications with a wide variety of goals can benefit the most from an application-aware self-management approach such as the one we are proposing. But in order to achieve application-aware self-management in VC systems we need to express the application itself as a parametric function. When expressing a multiscale application parametrically, we can define it as a series of functions whose input, or domain, is given by the application *parameters* and the output, or codomain, is given by one application *metric* per function; each metric is a quantitative measure relevant to the application goals. For example, for a Molecular Dynamics simulation, its parameters include type of solvent, number of cooling steps, number of heating steps, and initial velocity, while its metrics include accuracy of the simulation, execution time, power consumption, and load balancing. Formally, a job in a VC application x_i is a tuple of N parameters:

$$x_i = (x_{i,1}, x_{i,2}, ..x_{i,N})$$

We denote $x_{i,j}$ as the jth parameter in the x_i tuple. When a job is collected, we obtain a tuple y_i of M metrics:

$$y_i = (y_{i,1}, y_{i,2}, ..y_{i,M})$$

Note that the user must define the type of parameters and the range in which the parameters vary:

$$x_{i,1} \in [Rlow_1, Rhigh_1], ..., x_{i,N} \in [Rlow_N, Rhigh_N]$$

The goal of our application-aware self-management framework is to find parameter values within these ranges to optimize one or more metrics, meet the application goals, and use computing resources as efficiently as possible; all of that without direct human intervention.

IV. KOTrees

Our comprehensive approach for self-management in VC applications incrementally builds an N-$dimensional$ tree for Knowledge Organization (KOTree) that explores the parameter space with the ultimate objective of optimizing the specific application goal. Our approach comprises building the KOTree data structure, online training, prediction of metrics, and exploration of parameter space at runtime.

A. Building the KOTree

KOTree is an organized data structure that can predict M *metrics* and can explore the N-$dimensional$ parameter space of an application effectively, while being built incrementally at runtime. We use N parameters of a job as coordinates to determine the path of the job in the KOTree. Using this path, we update statistics of the M metrics stored in tree nodes along the path. Note that each node is not a job, but it contains statistical information on a bundle of jobs that are close in the parameter space. In particular, each node includes counters such as number of jobs that have contributed to the node (k), minimum value observed (Min), maximum value observed (Max), and other information that allow us to compute statistics such as expected value (E) and variance (S). To compute the statistics we use the Welford algorithm [26], [27] as follows: In the first step, expected value and variance are initialized as $E_1 = y_1$ and $S_1 = 0$, where y_1 are the metrics of the first job. For subsequent jobs, we use the recurrence formulas in Equation 1 to update E and S.

$$\begin{aligned} E_k &= E_{k-1} + \frac{(y_k - E_{k-1})}{k} \\ S_k &= S_{k-1} + (y_k - E_{k-1}) * (y_k - E_k) \end{aligned} \quad (1)$$

For a node with 2 or more jobs contributing to it, the k^{th} estimate of the variance is $s^2 = \frac{S_k}{(k-1)}$, E_k is the kth estimate of the mean, and S_k is the kth estimate of the standard deviation.

We selected the Welford algorithm because it enables the accurate calculation of the running variance. That is, the method computes the variance as the jobs arrive one at a time and the jobs do not need to be saved. Thus, KOTree is an one-pass algorithm and does not need to store the dataset, meeting the requirements of stream mining and making it more efficient than lazy-learning methods. For a KOTree with N dimensions and height ρ [1], the maximum number of nodes is $O(2^{N\rho})$. Note that the height of the tree is predefined by the user and remains constant during the whole prediction and exploration process.

The way we determine the path to which a job x_i contributes information to, is by mapping the job to an *n-gram* [2] of length ρ which we refer to as a key. The mapping is as follows: we define an initial N-$dimensional$ space of parameter ranges (i.e., $[Rlow_1, Rhigh_1]$, $[Rlow_2, Rhigh_2]$, ..., $[Rlow_N, Rhigh_N]$). Then, we divide the initial space into 2^N hypercubes of the same size. Every hypercube is given an unique identifier ranging from 0 to $2^N - 1$ based on its coordinates in the space. Every parameter j of a new job x_i is evaluated by a function $\delta(x_{i,j})$; this function outputs 0 if its value $x_{i,j} < Rlow_j + 0.5(Rhigh_j - Rlow_j)$, and 1 otherwise. We use Equation 2 to determine the identifier of the first hypercube (id_{hc1}), or node of depth 1, to which x_i belongs.

$$hypercube\_id(x_i) = id_{hc1} = \sum_{j=1}^{N} 2^{j-1}\delta(x_{i,j}) \quad (2)$$

The initialization of the first hypercube is the initial step in assigning a *key* to a job x_i; we continue assigning hypercubes recursively up to the specified height of the tree (ρ). At each level we update the parameter ranges $[Rlow_1, Rhigh_1]$, $[Rlow_2, Rhigh_2]$, ..., $[Rlow_N, Rhigh_N]$ based on the current partition of the N-dimensional space. The final key is formed by concatenating the ρ hypercube identifiers as follows:

$$key = "id_{hc1} id_{hc2} ... id_{hc\rho}"$$

To further illustrate the mapping of jobs into $keys$, Figure 1 shows the way in which a job x_i with 3 parameters is assigned to one of 8 hypercubes (top); Figure 1 also depicts the recursive procedure to determine the key for x_i by forming an octree (bottom).

A job is mapped to a single key, but a key can be associated to multiple jobs. The mapping we use to associate a job to a key assumes that a uniform partition of the space can accurately capture optimal parameter variability. This may not always be the case, but we knowingly trade accuracy for performance. For future work we are examining different types of mapping. For a better understanding lets assume a VC application with three parameters all ranging between 0 and 1 (i.e., $\forall j : Rlow_j = 0 and Rhigh_j = 1$) and two metrics, i.e.,

[1]The height of a tree is given by the length of the path from the root to the deepest node.

[2]In probability *n-gram* is a contiguous sequence of *n* symbols from a given sequence of text.

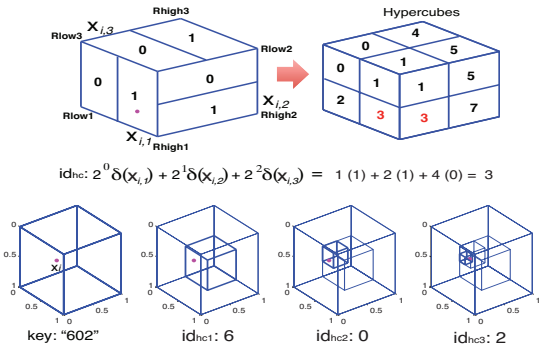

Figure 1. Key generation for x_i in a 3-$dimensional$ space.

CPU time and error. Figure 2 shows a partial KOTree with 2 branches for this application. The first branch comprises the $root$ and nodes $n0$, $n03$, and $n035$; the second branch comprises the $root$ and nodes $n6$, $n60$, and $n602$. Every node saves the mean (E), variance (S), number of jobs that contributed information to the node (k), minimum (min) and maximum (max) values observed for a single metric (e.g., CPU time). For didactic purposes, we used 2 boxes per node, one showing the *time metric* and its statistical values and one with the *error metric*. For the sake of readability of the example, we do not show the actual statistics for the error metric.

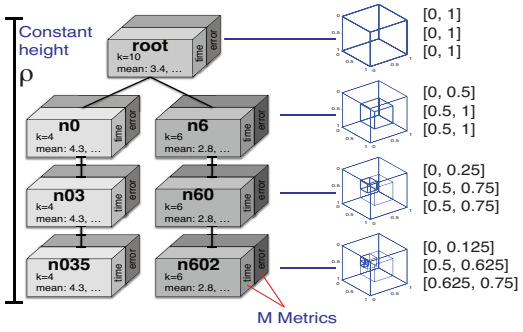

Figure 2. Fragment of a tree showing the knowledge organization and representation for an application with 3 parameters and 2 metrics.

B. Online training

In order to use our KOTree for prediction of metrics and exploration of parameter spaces, a KOTree is trained at runtime using metrics of collected jobs to update statistics of nodes in the tree. The training process starts with an empty KOTree and begins when we collect metrics y_1 for the first job x_1. Each new job x_i is associated to its metrics tuple y_i, which is feed to the KOTree.

Figure 3 shows the process of updating a tree at runtime using the information of a new job x_i. In this example, each node in the tree saves two metrics: CPU time ($time$) and accuracy of the solution ($error$). For each of the two metrics a node stores its statistical information (e.g.,

number of jobs that contributed to the node k, minimum observed value, maximum observed value, statistical mean and variance). The tree is the same one we used in the previous example and was constructed using 10 jobs. Recall that x_i represents a job that was executed using N parameters $x_i = \langle x_{i,1}, x_{i,2}, ..., x_{i,N} \rangle$ and whose execution can be measured by M metrics $y_i = \langle y_{i,1}, y_{i,2}, ..., y_{i,M} \rangle$. In this specific example y_i is $\langle y_{i,time}, y_{i,error} \rangle$. When a new job x_i with e.g., parameters $\langle 0.1, 0.7, 0.8 \rangle$ arrives, we compute its *key* = "id_1" . "id_2" . "id_3" = 642, where

$id_1 = 1(0) + 2(1) + 4(1) = 6 \quad [0,1], [0,1], [0,1]$
$id_2 = 1(0) + 2(0) + 4(1) = 4 \quad [0,0.5], [0.5,1], [0.5,1]$
$id_3 = 1(0) + 2(1) + 4(0) = 2 \quad [0,0.25], [0.5,0.75], [0.75,1]$

and the ranges to the right represent the updated set of ranges for the specific hypercube. The key 642 means that the new job x_i will contribute information to the $root$ and nodes $n6$, $n64$, and $n642$. In Figure 3.b we show the modifications to the tree after adding the new job. First we update the observed values in $root$ (e.g., instead of $k = 10$ now $k = 11$ and minimum time is 1.6 hours), then we update mean and variance for all the different metrics, in this case time and error, using the Welford algorithm in Equation 1. We repeat the updating process for node $n6$. Finally, we add nodes $n64$ and $n642$ which did not exist up to this point. Since

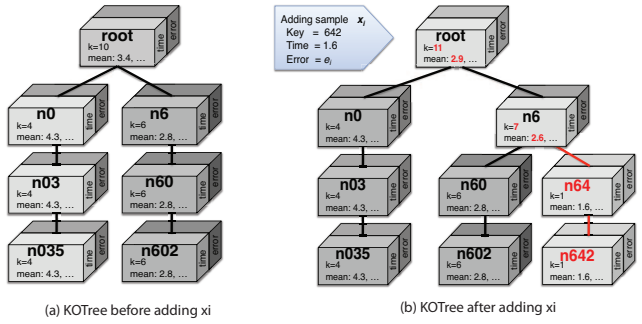

Figure 3. Online training of KOTree: (a) shows a KOTree before adding a new job x_i, and (b) shows the new KOTree after adding x_i.

for every update we just have to traverse one path of length ρ, the computational complexity of updating the KOTree is $O(M * \rho + 1)$; because ρ and M are defined in advance and never change, the complexity of updating the tree at runtime is $\approx O(1)$.

When we consider metrics with a dynamic behavior that change over time with, for example type of resources, we can use an asymptote to decrease the relevance of old metrics. This function replaces the actual number of jobs k that contributed information to a node, to a new value $\hat{k}$ as shown in Equation 3, where Δt is the number of time units since the last time the node was updated. For VC systems, a time unit is given in hours and the update interval may comprise 1 day. For systems with a faster or slower pace we can change the time unit and the update interval accordingly.

$$\hat{k} = \left\lceil \frac{k}{(\Delta t/24) + 1} \right\rceil \tag{3}$$

C. Prediction of metrics

The length of multiscale parametric applications is hard to estimate. Thus, prediction is a crucial step of our KOTreee algorithm. When it is time to generate a new job, we use a trained KOTree to predict the job length. More generally, we traverse a trained KOTree following the path associated to the new job and predict its expected metrics. We call a *metric of interest* the specific metric over which we perform a prediction (e.g., a metric of interest can be flops). To make a prediction using a KOTree, given a new job $\hat{x}_i$ we need to:

- Calculate the *key* for the new job $\hat{x}_i$
- Traverse the nodes in the path given by the ρ n-grams in $\hat{x}_i$'s *key*. For example, given key_i ='603', $Path_i = \{root, n6, n60, n603\}$).
- For each node in the path, keep track of its expected value E and variance S.
- Find the node with lower variance that is larger than 0.
- Use the expected value E of the metric of interest in the node to make the prediction for $\hat{x}_i$, ($\hat{x}_i \rightarrow \hat{y}_i = E$)

Figure 4 shows how our method makes a prediction for the new job $\hat{x}_i$. Lets assume the new job has parameters $(0.2, 0.7, 0.6)$ and we want to predict its expected CPU time. We first calculate the key for $\hat{x}_i$, e.g., key_i ='603', and $Path_i = \{root, n6, n60, n603\}$. Then, we traverse the path in Figure 4. Note that $Path_i$ includes node $n603$ that does not exist in the KOTree at this point in time. Thus, for each of the nodes in $Path_i$, except for $n603$, we compare their statistics and select the node with smallest variance, which in this case it is node $n60$. Finally, we emit the expected value E from $n60$ as the predicted CPU time for the job $\hat{x}_i$ as $\hat{y}_i = 3.0$. Note that in this example we assume we are interested in the expected CPU time, but in the same way we can make predictions for the minimum CPU time, maximum CPU time, or variance of CPU time.

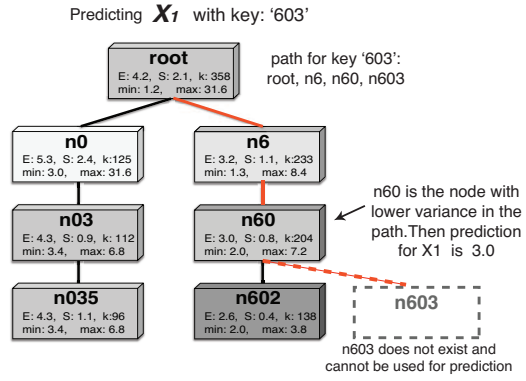
Predicting X1 with key: '603'
root
E: 4.2, S: 2.1, k: 358
min: 1.2, max: 31.6
path for key '603':
root, n6, n60, n603
n0
E: 5.3, S: 2.4, k:125
min: 3.0, max: 31.6
n6
E: 3.2, S: 1.1, k:233
min: 1.3, max: 8.4
n03
E: 4.3, S: 0.9, k: 112
min: 3.4, max: 6.8
n60
E: 3.0, S: 0.8, k:204
min: 2.0, max: 7.2
n60 is the node with lower variance in the path.Then prediction for X1 is 3.0
n035
E: 4.3, S: 1.1, k:96
min: 3.4, max: 6.8
n602
E: 2.6, S: 0.4, k: 138
min: 2.0, max: 3.8
n603
n603 does not exist and cannot be used for prediction

Figure 4. Fragment of a tree showing the path for predicting the CPU time (i.e., *metric of interest*) of a new job with key 603.

Similarly to when we update a KOTree, the computational cost of making a prediction for a tree of height ρ is $O(\rho+1)$. Since ρ is defined in advance and never changes, the complexity of making a prediction is constant $\approx O(1)$.

D. Exploration of the N-dimensional space of parameters

The key question still unanswered is how to generate parameters for a new job x_i. Given a multiscale application with a well-defined set of N parameters, KOTree can explore this N-$dimensional$ space efficiently and find regions where parameters are more likely to produce jobs whose metrics align with the application goal. To achieve this objective, we enforce a partial ordering of the nodes and explore them proportionally to their expected gain with respect to the application goal (app_{goal}). For example, if the app_{goal} is minimizing the average execution time of a batch of jobs, then we order the nodes with respect to their expected value associated to the metric of interest CPU time. A reduced list of nodes with the smallest expected CPU time is used to generate parameters for new jobs.

For each node in the list, we generate jobs whose parameters fall within the specific parameter ranges of the node proportionally to their relative gain with respect to the application goal. Figure 5 shows an example for which the list consists of 5 nodes, 2 of them have expected CPU time of 1 hour ($n6, n657$) and the other 3 nodes have expected CPU time from 1.2 hrs to 1.6 hrs ($n1, n62, n120$). For didactic purposes, we assume a generation of 100 jobs. Since the goal of this application is minimizing CPU time, the smaller their CPU time, the more jobs are generated for each node. In this example, KOTree generates 24 jobs for each of the 2 nodes with $Expected(CPUtime) = 1.0$, 20 jobs for the node with $Expected(CPUtime) = 1.2$, 17 jobs for the node with $Expected(CPUtime) = 1.4$, and 15 jobs the node with $Expected(CPUtime) = 1.6$. We provide more details on parameter exploration in Section V.

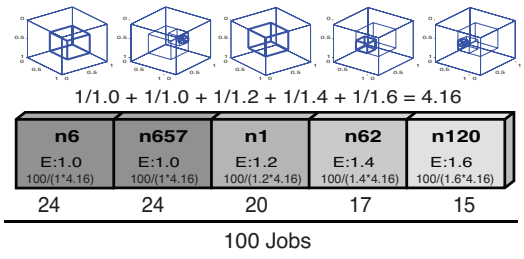

Figure 5. Sorted list of nodes to be explored for the generation of new jobs.

Once it is determined that a new job will be generated for a specific hypercube, we generate the job N parameters within the given hypercube boundaries. There are three driving factors for generating values of the parameters: exploration of the node space, exploitation of the best solution, and sampling efficiently closer to the hypercubes boundaries [28], [29]. For exploring the node space we generate a random number in the parameter range of the specific node by using a uniform distribution. For exploiting the best solutions, we use the top solution only if its value is in the parameter range of the node, and a random value otherwise. For sampling efficiently closer to the hypercube boundaries, and mitigate possible inaccuracy due to the uniform partitioning, we generate the *i-th* Chebyshev node [29] using the total number of jobs that have to be generated for this node. KOTree uses a combination of the three with an equal probability.

E. Overcoming KOTree limitations

In KOTree, the ranges of N parameters need to be known in advance. However, for some real-world applications it is impossible to estimate these ranges with total certainty. This problem can be addressed by allowing a bottom-up expansion of the tree. That is, by moving the root one level up and adding an intermediate node at height 1. The process of updating an existing KOTree is as follows: assume that we already built a KOTree of height $\rho = 3$ with ranges $[Rlow_1, Rhigh_1]$, $[Rlow_2, Rhigh_2]$, ..., $[Rlow_N, Rhigh_N]$, and there is a new job x_i with N parameters which falls outside one or more of the ranges specified originally, where $x_{i,j}$ denotes the parameter j of x_i outside the range. We can add an intermediate node at height 1 and calculate its key using Equation 4, where $\delta'(x_{i,j})$ is a function that outputs 1 if $x_{i,j} < Rlow_j + 0.5(Rhigh_j - Rlow_j)$, and 0 otherwise.

$$hypercube\_id(x_i) = id_{hc1} = \sum_{j=1}^{N} 2^{j-1}\delta'(x_{i,j}) \qquad (4)$$

Once we compute a key and insert the intermediate node, the key is added as a suffix for all nodes in the original tree. We update the KOTree height ρ to $\rho + 1$ and KOTree ranges are adjusted as follows:

$$Update \begin{cases} Rlow_j \leftarrow 2Rlow_j - Rhigh_j, & \text{if } \delta'(x_{i,j}) = 1 \\ Rhigh_j \leftarrow 2Rhigh_j - Rlow_j, & \text{if } \delta'(x_{i,j}) = 0 \end{cases}$$

V. Modular framework

To provide application-aware self-management in VC we integrate KOTree as the learning engine of a modular framework. Specifically, the framework is composed by the learning engine and three modules (i.e., job generation, resource assessment, and result collection). The different modules are used as software interfaces to provide a common communication protocol that can be understood by our KOTree regardless of the computational platform (e.g., VC, Grid, Cloud) and the actual scientific application it is optimizing.

The *learning engine* uses KOTree to explore the parameter space and to generate jobs likely to optimize application-specific goals in the amount that is better suited for the available resources. It also makes predictions with respect to new jobs and acts as a coordinator among the other three modules.

The *job generation module* provides the application with specific parameters for each new job and the number of replicas to be executed. Job parameters are actually chosen by the KOTree learning engine. The function of this module is to communicate job parameters in a format that is understandable by the application. This module outputs an XML file that can be parsed by any application to generate actual jobs.

The *resource assessment module* determines the capacity of available resources within the past predetermined unit of time

(seconds, minutes, hours). As described in Estrada *et al.* [30], a VC system can be characterized by its different latency phases, distribution, and starvation patterns. Thus, in this module we determine the expected amount of flops that the system can successfully process. This module receives three log files from the distributed system: one with the log of requests and jobs assigned, one with temporal information per collected job, and one with the characteristics of jobs that are assumed to be timed-out by the system. The module passes the amount of estimated flops that the system can handle to the learning engine.

The *result collection module* parses the collected results, extracts metrics for each job, and returns the metrics and their associated combination of parameters to the KOTree learning engine. The output stream of this module is used by the KOTree learning engine for online training.

Figure 6 depicts the structure of our application-aware self-managed framework in the context of a VC system. The module interaction can be broken down into two main processes: (a) exploration and prediction and (b) online training.

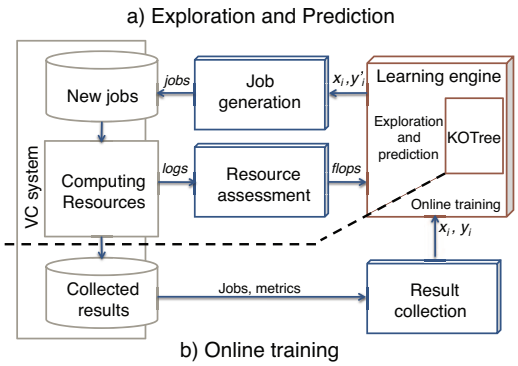

Figure 6. Modular framework

The process of exploration and prediction includes assessing the system capability and using KOTree to select parameters that can be used to generate new jobs as illustrated in Algorithm 1. This process involves communication and coordination among the resource assessment ($RESOURCE\_ASSESMENT$), the learning engine ($LEARNING\_ENGINE$), and the job generation ($JOB\_GENERATION$) modules. The resource assessment estimates the current computational capability of the system based on its immediate history, the learning engine uses KOTree to determine the best parameters to be explored, and the job generation communicates the job features to the application. Recall that for performance sake, the learning engine keeps track of a sorted list of the best candidate nodes ($nodes^{\beta}$) to be explored simultaneously. We use the app_{goal} as the scoring function for each node and sorting criterion for the list. The user defines the size of the list, which can be any value up to the total of nodes in KOTree, although smaller lists are more efficient. Even though the size of the list is static, this does not critically impact the accuracy of the algorithm because the nodes in the list change as new jobs are collected. Note that $node_r^{\beta}$ is the *r-th* node in $nodes^{\beta}$.

Algorithm 1 Exploration and Prediction

$\underline{RESOURCE\_ASSESMENT}$:
Define t as the current unit of time
$\alpha_1 \leftarrow 85th$ quantile of distributed jobs (in flops) for $t-1$
$\alpha_2 \leftarrow 85th$ quantile of distributed jobs (in flops) whose execution latency has exceeded a time-out bound in $t-1$
$\alpha_3 \leftarrow$ number of unsatisfied requests in $t-1$
$\alpha_4 \leftarrow$ average assigned workload per request (in flops) in $t-1$
$\alpha_{flops} \leftarrow \alpha_1 - \alpha_2 + (\alpha_3 * \alpha_4)$
Pass $\alpha_{flops} \longrightarrow LEARNING\_ENGINE$

$\underline{LEARNING\_ENGINE}$:
for each $node_r^{\beta} \in nodes^{\beta}$ **do**
  Assign a weight $0 < w_r < 1$ to $node_r^{\beta}$ based on the app_{goal} improvement with respect to the rest of the nodes in $nodes^{\beta}$ such that $\sum w_r = 1$
  Generate jobs in w_r proportion until the flops' quota $w_r * \alpha_{flops}$ is fulfilled.
  for each new job $x_i \in node_r^{\beta}$ **do**
    Generate its specific parameters within the N ranges in $node_r^{\beta}$ given one of three criteria (exploration, exploitation, Chebyshev)
    Predict $\hat{y}_i$ as expected flops for x_i using $KOTree$
    Pass $x_i, \hat{y}_i \longrightarrow JOB\_GENERATION$
  end for
end for

$\underline{JOB\_GENERATION}$:
for each new job x_i **do**
  Output a standard XML file with its parameters and estimated flops $\hat{y}_i$ to be used by the application to generate new jobs
end for

The process of online training takes place after executed jobs are collected. The learning engine uses this information to update the model in KOTree and improve its ability to make predictions. This process is shown in Algorithm 2 and requires communication between application, result collection module ($RESULT\_COLLECTION$) and learning engine.

Algorithm 2 Online training

$\underline{RESULT\_COLLECTION}$:
for each collected job with original parameters x_i and with actual flops y_i returned by the application **do**
  Pass $x_i, y_i \longrightarrow LEARNING\_ENGINE$
end for

$\underline{LEARNING\_ENGINE}$:
Define $flops$ as the metric of interest
Get x_i, y_i from *RESULT_EVALUATION*
for each job x_i with actual flops y_i **do**
  Calculate key_i for x_i
  for each $node_k \in KOTree$ in the path of key_i **do**
    Use y_i to update statistics in $node_k$ for metric $flops$
    if Expected $flops \in node_k < max(flops \in nodes^{\beta})$ **then**
      Save $node_k$ in the sorted list $nodes^{\beta}$
    end if
  end for
end for

VI. EVALUATION

A. Application settings

To evaluate our framework in a simulated VC environment, we define three case studies corresponding to three types of applications (as explained in Section III) with two metrics of interest (i.e., CPU time and error of the solution) and different goals:

- *Case Study 1* assumes a latency-oriented application with the goal of minimizing expected CPU time per job.
- *Case Study 2* assumes an accuracy-oriented application with the goal of increasing accuracy of results regardless of the time it takes to compute them.
- *Case Study 3* assumes a convergence-oriented application with the goal of finding a balance between faster jobs and accurate results.

We evaluate the three case studies over a set of 14 scenarios corresponding to specific implementations of each application, with one to 4 parameters (i.e., $p1, p2, p3, p4$) and with two *metrics* (i.e., $time$ and $error$). Each of the 14 scenarios is built upon four basic blocks. Table I shows these blocks for $p1$, $p2$, $p3$, and $p4$. The reminder 10 scenarios ($p1p2$, $p1p3$, $p1p4$, $p2p3$, $p3p4$, $p2p4$, $p1p2p3$, $p1p2p4$, $p2p3p4$, $p1p2p3p4$) are built using linear combinations for 2, 3, and 4 parameters (e.g., $f_t(p1, p2) = f_t(p1) + f_t(p2)$, $f_e(p1, p2, p4) = f_e(p1) + f_e(p2) + f_e(p4)$). For the building blocks, we chose a total of 8 functions (2 per scenario) with a variety of characteristics, ranging from oscillatory to monotonic. When combined, these functions create a wide range of optimization landscapes (some of them with multiple local minima, some with very well-defined valleys and global minima, and some with pronounced and semi-random spikes) and allow us to prove the robustness of our KOTree algorithm under very different conditions. Note that this set of functions provides us with scenarios where similar parameters may or may not produce similar metrics. This is an important feature that allows us to test whether the uniform partition method represents a critical drawback of our KOTree algorithm.

Scenario	Time function ($f_t \rightarrow$ *time metric*)
$p1$	$f_t(p1) = 2 + 10(\sin(p1^3)e^{-p1})$
$p2$	$f_t(p2) = \log(p2)\sin(8p2) + 2$
$p3$	$f_t(p3) = 0.5 + \sin(p3)\cos\left(\frac{1}{p3+1}\right) + 1$
$p4$	$f_t(p4) = 2\log(p4 + 1)$
Scenario	**Error function ($f_e \rightarrow$ *error metric*)**
$p1$	$f_e(p1) = 0.6 + \sin(2p1)\frac{\arctan(p1)}{2}$
$p2$	$f_e(p2) = 1 + \frac{\sin(2^{p2})}{\log(p2+1)}$
$p3$	$f_e(p3) = 0.5 + \frac{\tan(p3)}{100}$
$p4$	$f_e(p4) = 2e^{-p4}2^{p4}$

Table I
DEFINITION OF 4 BUILDING BLOCKS FOR SIMULATING SPECIFIC IMPLEMENTATIONS OF A MULTISCALE PARAMETRIC APPLICATION

To compare and contrast the effectiveness of our KOTree algorithm, we additionally consider two non-KOTree testing methods for generation of job parameters: random (RND) and simulated annealing (SAN). We also define two KOTree variants that use KOTree in slightly different ways (i.e., KOM, KOE). Following is the description of these four testing methods:

RND Generation of parameters per job using a random value within specified ranges per parameter. Prediction of job size is made using the average job size over the jobs successfully completed.

SAN Generation of parameters per job using a simulated annealing approach, keeping track of the best solution for the associated application goal, and producing variations of parameters in the direction with better gain given the probabilities of the algorithm. Prediction of job size is made using a weighted sum considering 0.5 job size of the best solution plus 0.5 the average job size of jobs successfully completed in the past 5 units of time.

KOM Generation of parameters per job using a KOTree driven by minimum values. The *M* in the acronym means that the list of nodes used to explore parameter space is sorted based on minimum values. Prediction of job size is made using the statistics for time in the KOTree structure as explained in Section IV-C.

KOE Generation of parameters per job using a KOTree driven by expected values. The list of nodes used to explore the parameter space is sorted based on expected values, thus the *E* in the acronym. Again, prediction of job size is made using the statistics for time in the KOTree structure.

B. Simulating multiscale applications with EmBOINC

In real multiscale applications, parameters are known ahead of time and they represent different aspects of the model or different scales of the application. However, the way in which a combination of parameters affects application performance and goals is unknown or hard to estimate. That is, the underlying functions can be unknown by the scientists. On the other hand, a simulated multiscale application can be represented as a combination of functions that react to the parameters in a more or less controlled way. In other words, in a simulated multiscale application it is possible to model how each parameter affects the application. For the sake of fairness, this information should not bias the overall results.

To address this challenge in a trustworthy way, we use EmBOINC [31] (our emulator of VC applications). In order for EmBOINC to include information about the multiscale application behavior, we add the capability to specify parameters and the way in which they affect application metrics. Given a specification provided by the user, our job generation module parses this specification and builds functions into a Perl module that is used at runtime to provide information of each job to EmBOINC. The user can define as many metrics and parameters as needed for each application. EmBOINC uses the application models only to produce job results, and returns a numeric value for each application metric through the job result message. Additionally, the flops metric is actually

bound to the execution of a job, which takes the amount of computation given by the specified model, plus delays associated to the platform (e.g., interruptions due to the non-dedicated nature of VC).

For all four testing methods we used BOINC (v.6.1.11), the default policies for generation, distribution, and validation of jobs. All the simulations presented in this paper ran using EmBOINC v.1.2, the same set of 12,470 hosts obtained from traces of Docking@Home, and accounted for the same number of simulated hours (168 hours, 1 week). For the experiments using KOTree, the height of the tree was set to 4. We empirically observed that even trees with a short height were able to considerably improve application goal and throughput, as we will discuss in Section VI-D. We tested three case studies, 14 scenarios, 4 algorithms per scenario, 5 simulations per algorithm, for a total of 840 simulations of BOINC applications.

C. Results

In this section we compare and contrast the different algorithms (i.e., random, simulated annealing, and the two variations of the KOTree algorithm) for automatic parameter generation in terms of (1) how well do they meet a specific application goal, and (2) how efficiently they use resources, expressing this efficiency in total throughput of the application over the same set of resources. Note that in the context of multiscale VC applications a poor choice of parameters results in applications that cannot meet their goal effectively and that do not use resources efficiently.

1) Application goal: To measure the likelihood that application goals are reached, we use the Kolmorgorov-Smirnov (KS) test [3] with 95% confidence to compare the empirical result distributions produced by the different testing algorithms [32]. We test the hypothesis $dist(X) < dist(SAN)$ where $dist(X)$ is the distribution produced by either the random ($dist(RND)$), KOTree minimum value ($dist(KOM)$), or KOTree expected value ($dist(KOE)$) algorithms respectively, and $dist(SAN)$ is the distribution produced by the simulated annealing algorithm. We use the simulated annealing distribution as the basis for the KS test because SAN performs considerably better than RND and thus, it is the strongest contestant against our KOTree algorithm. Figure 7 presents the KS test $dist(X) < dist(SAN)$, for the three case studies, where X is RND, KOM, and KOE respectively. The black bars represent the percentage of scenarios where the alternate algorithms (i.e., RND, KOM, and KOE) are statistically smaller than SAN (better than SAN), the gray bars represent cases where the alternate algorithms are similar to SAN, and the white bars represent cases where the alternate algorithms are larger to SAN (worse than SAN).

In *Case Study 1*, for the random algorithm $RAND$ we observe that in only one case RND is similar to SAN and worse in the remainder 13 cases. For the KOM algorithm we observe that in 57% of the cases (8 out of 14) KOM outperforms SAN, in one case both algorithms behave similarly ($p1p2p3p4$), and in 36% of the cases (5 out of 14) SAN is better than KOM ($p2, p4, p1p2, p2p3, p2p4$). For the KOE algorithm we observe that in 64% of the cases (9 out of 14) KOM outperforms SAN, in two cases both algorithms behave similarly ($p2$ and $p1p3$), and in 22% of the cases (3 out of 14) SAN is better than KOE ($p4, p1p2p3, p1p2p4$).

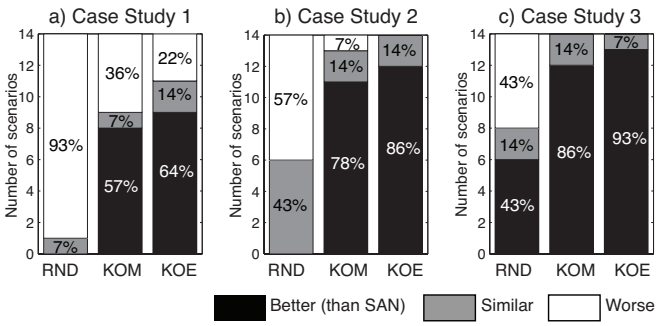

Figure 7. Kolmorgorov-Smirnov test for $dist(X) < dist(SAN)$, where X is RND, KOM, and KOE respectively.

In *Case Study 2*, for the random algorithm $RAND$ we observe that in 43% of the cases (6 out of 14) RND is similar to SAN and worse in the remainder 8 cases. For the KOM algorithm, we observe that in 78% of the cases (11 out of 14) KOM outperforms SAN, in two cases both algorithms behave similarly ($p2, p4$), and in only 7% of the cases (1 out of 14) SAN is better than KOM ($p1p2p4$). For the KOE algorithm we observe that in 86% of the cases (12 out of 14) KOM outperforms SAN, in the reminder two cases both algorithms behave similarly ($p2$ and $p4$), and in none of the cases SAN is better than KOE.

In *Case Study 3*, for the random algorithm $RAND$ we observe that in 43% (6 out of 14) of the cases RND is better than SAN, they are similar in two cases ($p4$ and $p3p4$), and worse in the other 43% (6 out of 14) of the cases. For the KOM algorithm we observe that in 86% of the cases (12 out of 14) KOM outperforms SAN and in the other two cases both algorithms behave similarly ($p4, p1p2$). In this case study SAN never outperforms KOM. For the KOE algorithm we observe that in 93% of the cases (13 out of 14) KOM outperforms SAN and only in one case both algorithms behave similarly ($p4$). Similarly to what happens for KOM, in this case study SAN never outperforms KOE.

2) Throughput: The experiments show an additional virtue of our KOTree-based approach. Since the two variations of the KOTree algorithm are able to predict the expected execution time of jobs in a fairly accurate way (in average doubling the accuracy with respect to RND and SAN), they also enable a better use of resources and ultimately increase throughput. This is possible because scheduling policies often rely on an accurate estimation of job size to perform their best assignments. Underestimating job sizes results in unrealistic expectations of resource usage and leads to wasting

[3]The KS statistic is the maximum difference between any pair of empirical distribution functions (EDFs).

resources due to replication of jobs falsely assumed to be timed-out. Overestimating job sizes provokes underutilization of resources. Figure 8 shows the normalized throughput with respect to the throughput produced by the simulated annealing algorithm (SAN). Our results indicate that KOM increases throughput with respect to SAN an average of 75%, 73%, and 85% for Case Study 1, Case Study 2, and Case Study 3 respectively. KOE increases throughput with respect to SAN an average of 132%, 61%, and 107% for Case Study 1, Case Study 2, and Case Study 3 respectively.

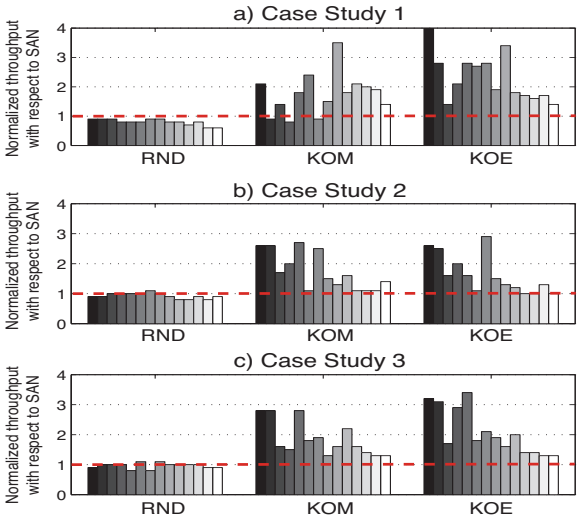

Figure 8. Normalized throughput with respect to SAN.

D. Discussion

Our empirical evaluation of three case studies simulating applications with different scientific goals and 14 scenarios representing different parametric implementations of the multiscale application, shows that KOTree is an effective mechanism to drive parameter optimization and effectively provides application-aware self-management.

The two variations of our KOTree algorithm (i.e., KOM and KOE) outperform a simulated annealing algorithm in 77% of the cases across application goals and application implementations. In 13% of the cases the performance is similar and only in 10% of the cases (9 out of 84) the simulated annealing algorithm outperforms our KOTree algorithms. We can group these 9 cases in 3 categories:

1) When the application has a single parameter with a small domain and the application has a well-defined global minimum for the metric of interest, we observe that the KOTree algorithm performs poorly compared to the simulated annealing algorithm.
2) When the application has only two parameters, one of them dominates the metric of interest and has a quasi-random behavior, KOM has lower performance than KOE. If we analyze the KOTrees produced by these scenarios, we observe that most nodes have a very large variance, indicating that the space partition was insufficient to cluster correctly the parameter space. From this set of experiments we can conclude that when the internal variance of nodes is large with respect to the minimum and expected values, then we should use KOE rather than KOM, or we need to use KOM with a larger tree height. In general, KOE performed consistently better than SAN for cases with large variances and one dominant parameter.
3) When the application relies on at least two parameters whose values do no cluster naturally and appear to be scatter all over the landscape of parameter values, we observe that any approach that organizes knowledge accordingly to the parameters proximity is not effective.

We also observe that accurate job length prediction is crucial for efficient resource utilization. Underestimating job lengths results in unrealistic expectations of resource usage and the possibility of wasting resources due to replication of jobs falsely assumed to be timed-out, while overestimation of job size can potentially cause underutilization of resources. In general, across the case studies and implementation scenarios analyzed in this paper, our KOTree algorithm is able to predict the expected length of new jobs fairly accurately, resulting in an average of 85% increased throughput and more than 340% increased throughput in 4 cases.

VII. Conclusions

In this paper we present an autonomic, modular framework for providing application-aware self-management for multi-scale applications running on VC. The learning engine of this framework is based on Knowledge Organization Trees (KOTrees). KOTree is a fully automatic method that (1) collects statistical information about how different parameter combinations affect one or more application metrics; (2) organizes gained knowledge in an N-$dimensional$ tree that can be efficiently searched and updated, enabling the possibility of runtime learning; (3) makes accurate predictions regarding one or more application metrics associated with an unexplored combination of parameters; and (4) provides an effective strategy to better explore the N-$dimensional$ space of parameters. At any point in time, we have an organized data structure that can predict multiple *metrics of interest* and explore the N-$dimensional$ space of parameters effectively.

Our empirical evaluation shows that our framework can effectively provide application-aware self-management in VC systems. Additionally, we observed that our KOTree algorithm is able to predict the expected length of new jobs accurately, resulting in an average of 85% increased throughput with respect to other algorithms.

Acknowledgment

This work was supported by the NSF IIS #0968350 entitled Collaborative Research: SoCS - ExSciTecH: An Interactive, Easy-to-Use Volunteer Computing System to Explore Science, Technology, and Health.

REFERENCES

[1] B. J. Adler and T. E. Wainwright. Studies in molecular dynamics. I. general method. *J. Chemical Physics*, 31:459, 1959.

[2] S. Hassan, D. Al-Jumeily, and A.J. Hussain. Autonomic computing paradigm to support system's development. In *Proc. of the International Conference on Development in eSystems Engineering*, 2009.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions On Systems Science And Cybernetics*, 4(2):100–107, 1968.

[4] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *J. of Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[5] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method (Wiley Series in Probability and Statistics)*. 2 edition.

[6] S. Guha, A. Meyerson, N. Mishra, and R. Motwani. Clustering data streams: Theory and practice. *IEEE TKDE*, 15:515–528, 2003.

[7] P. L. Bartlett, S. Ben-David, and S. R. Kulkarni. Learning changing concepts by exploiting the structure of change. *Machine Learning*, 41(2):153–174, 2000.

[8] P. Zhang, B. J. Gao, X. Zhu, and L. Guo. Enabling fast lazy learning for data streams. *IEEE International Conference on Data Mining*, 0:932–941, 2011.

[9] H. Yang, H. Liu, and J. He. DELAY: a lazy approach for mining frequent patterns over high speed data streams. In *Proc. of the 3rd international conference on Advanced Data Mining and Applications (ADMA07)*, 2007.

[10] K. Ueno, X. Xi, E. Keogh, and D.-J. Lee. Anytime classification using the nearest neighbor algorithm with applications to stream mining. In *Proc of the 6th International Conference on Data Mining (ICDM '06)*, 2006.

[11] S.Z. Erdogan, T.T Bilgin, and J. Cho. Fall detection by using K-nearest neighbor algorithm on WSN data. In *Proc. of the IEEE GLOBECOM Workshops (GC Wkshps)*, 2010.

[12] H. He, S. Chen, K. Li, and X. Xu. Incremental learning from stream data. *IEEE Transactions on Neural Networks*, 22(12):1901 –1914, 2011.

[13] Z. Leng, C. Fu, and X. Gu. Stock price index prediction based on mobile data mining. In *Proc. of the International Conference on E-Business and E-Government (ICEE)*, 2010.

[14] B. Raahemi, A. Kouznetsov, A. Hayajneh, and P. Rabinovitch. Classification of Peer-to-Peer traffic using incremental neural networks (Fuzzy ARTMAP). In *Proc. of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2008.

[15] H. Kawashima, R. R. Sato, and H. Kitagawa. Models and issues on probabilistic data streams with Bayesian Networks. In *Proc. of the International Symposium on Applications and the Internet (SAINT)*, 2008.

[16] X. Qing, C. Bo-Wei, Z. Chang-Wei, Y. Ping-Gang, and L. Yong-Hong. Study on application of Bayesian classifier model in data stream. In *Proc. of the International Conference on Computational and Information Sciences (ICCIS)*, 2010.

[17] F. Al Machot, A. H. Mosa, K. Dabbour, A. Fasih, C. Schwarzlmuller, M. Ali, and K. Kyamakya. A novel real-time emotion detection system from audio streams based on bayesian quadratic discriminate classifier for ADAS. In *Proc. of the 16th International Symposium on Theoretical Electrical Engineering Nonlinear Dynamics and Synchronization (INDS)*, 2011.

[18] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the International Conference on Knowledge Discovery and Data*, 2000.

[19] J. Yang, H. Chen, S. Hariri, and M. Parashar. Autonomic runtime manager for adaptive distributed applications. In *Proc. of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, 2005.

[20] L. R. Moore, Jr., M. Kopala, T. Mielke, M. Krusmark, and K. A. Gluck. Simultaneous performance exploration and optimized search with volunteer computing. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[21] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[22] J. Holger and B.A. Pletsch. Exploiting large-scale correlations to detect continuous gravitational waves. *Physical Review Letters*, 103(1):1–4, 2009.

[23] D. J. Rowlands, D. J. Frame, D. Ackerley, T. Aina, B. B. Booth, C. Christensen, M. Collins, N. Faull, C. E. Forest, B. S. Grandey, E. Gryspeerdt, E. J. Highwood, W. J. Ingram, S. Knight, A. Lopez, M. Massey, and F. McNamara. Broad range of 2050 warming from an observationally constrained large climate model ensemble. *Nature Geoscience*, 5:256 – 260, 2012.

[24] M. Taufer, M. Crowley, D. Price, A. Chien, and C. L. Brooks III. Study of an accurate and fast protein-ligand docking algorithm based on molecular dynamics. *Concurrency and Computation: Practice and Experience*, 17(14):1627–1641, 2005.

[25] E. Cochran, J. F. Lawrence, C. Christensen, and A. Chung. A novel strong-motion seismic network for community participation in earthquake monitoring. *IEEE Instrumentation and Measurement Magazine*, 12(8), 2009.

[26] D. E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, 1998.

[27] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4:419–420, 1962.

[28] G. James. Exploration and exploitation in organizational learning. *Organization Science*, 2(1):71–87, 1991.

[29] D. C. Montgomery and G. C. Runger. *Applied Statistics and probability for engineers*, chapter Elements of Statistics II: Inferential Statistics. McGraw-Hill, 2005.

[30] T. Estrada, M. Taufer, and K. Reed. Modeling job lifespan delays in volunteer computing projects. In *Proc. of the 9th IEEE International Symposium on Cluster Computing and Grid (CCGrid)*, 2009.

[31] T. Estrada, M. Taufer, and D. P. Anderson. Performance prediction and analysis of boinc projects: An empirical study with emboinc. *J. Grid Computing*, 7(4):537–554, 2009.

[32] R. H. C. Lopes, I. Reid, and P. R. Hobson. The two-dimensional Kolmogorov-Smirnov test. In *Proc. of the International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, 2007.